Object-Oriented Macromedia Flash MX

WILLIAM DROL

Apress™

ISBN (pbk): 1-59059-014-7

Printed and bound in the United States of America 12345678910

Technical Reviewer: Thomas Burr

Editorial Directors: Dan Appleman, Peter Blackburn, Gary Cornell, Jason Gilmore, Karen Watterson, John Zukowski

Managing Editor: Grace Wong

Copy Editor: Nicole LeClerc

Compositor: Susan Glinert

Artist and Cover Designer: Kurt Krames

Indexer: Valerie Haynes Perry

Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email `orders@springer-ny.com`, or visit `http://www.springer-ny.com`.

Outside the United States, fax +49 6221 345229, email `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 9th Street, Suite 219, Berkeley, CA 94710.

Phone 510-549-5930, fax: 510-549-5939, email `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 2

Introduction to OOP

Object-oriented programming (OOP) is an overused term for a simple idea. OOP is not a computer language or any kind of software application. OOP is a commonsense way to examine problems and break them down into smaller pieces. These pieces are *objects,* the building blocks of software.

Not long ago, developers had to write large, complex applications using countless lines of computer code. These applications described exactly what the computer needed to do in a painful, systematic fashion. Often, the code was heavily interdependent. If anything changed, a whole series of bugs might appear, sometimes making it nearly impossible to locate the true source of the errors.

Frequently, software patches attempted to bypass errors without addressing the real problem. If the project found itself in a tailspin, the only choice was to abandon large portions of the code and start over. The rebuilding process usually suffered the same fate, producing a purgatory for coding madness.

In an attempt to stop these problems, OOP was born. OOP is not perfect, but it offers a simple way to handle complex problems. It's also a commonsense approach. Think of your car. It's an object-oriented system formed by many discrete pieces (or objects). The car itself is an object. The engine inside the car is an object. The exhaust system is an object that contains even more objects. The engine depends upon the exhaust system, but it has no idea how the exhaust system works. Fortunately, your car parts conspire so that you may drive down to the corner store and buy a frozen treat.

The Scoop on OOP

When you get into your car, you turn the key, the car starts, and off you go. You don't need to understand how the car parts work to find yourself in rush-hour traffic. The car starts when you turn the key. Car designers hide the messy internal details so you can concentrate on important things like finding another radio station. OOP calls this concept *encapsulation.*

Are you old enough to remember fuel stations before the self-service era? You could drive into these places and somebody *else* would fill up your tank. The station attendant knew about OOP long before you did. He put the fuel nozzle into the tank (any tank) and pumped the fuel! It didn't matter if you drove a Ford, a

Chrysler, or a Datsun. All cars have fuel tanks, so this behavior is easy to repeat for any car. OOP calls this concept *polymorphism.*

Do you need polymorphic behavior in OOP? Yes, you might need to create things that bounce, for instance. A ball bounces in a normal ball-like fashion, a squeaky toy bounces in an unpredictable fashion, and a bowling ball doesn't bounce much at all. The point is, they can all respond to the same command: BOUNCE! This process works very well, especially when you don't know (in advance) which objects you may need to control.

The last major part of OOP is inheritance. *Inheritance* allows you to build new applications based on pieces and parts of existing applications. You start with an existing piece of computer code and then introduce a desirable new feature. The result is a new piece of computer code with qualities identical to the original, but with added functionality. If you do this carefully, you can stop building code from scratch.

Now that you have an overview, I will focus on classes, objects, and properties. These are the OOP power tools that solve real problems.

Classes and Objects

There's a subtle difference between a class and an object. A *class* is a self-contained description for a set of related services and data. Classes list the services they provide without revealing how they work internally. You cannot do any real work with a class; you can only describe what the class may offer. To do work, you need an object.

Suppose you want to build a house. Unless you build it yourself, you need an architect and a builder. The architect drafts a blueprint, and the builder uses it to construct your house. Software developers are architects, and classes are their blueprints. You cannot use a class directly, any more than you could move your family into a blueprint. Classes only describe the final product. To actually do something you need an *object.*

If a class is a blueprint, then an object is a house. Builders create houses from blueprints; OOP creates objects from classes. OOP is efficient. You write the class once and create as many objects as needed.

Properties

Properties give individual objects unique qualities. Without properties, each house (from the previous example) would remain identical to its neighbors (all constructed from the same blueprint). With properties, each house is unique, from its exterior color to the style of its windows. These are properties.

In OOP, you write classes to offer predefined behaviors and maybe hold some data. Next, you create one or more objects from a class. Finally, you endow objects with their own individual property values. The progression from classes to objects to objects with unique properties is the essence of OOP.

Understanding Basic OOP Concepts

Encapsulation, polymorphism, inheritance . . . they're just words. Don't memorize them just because you think must. Understand the concepts first; remember the names later. There's a big difference between knowing the name of something and knowing something. One of Richard Feyman's stories about a lesson his father taught him applies here:

> *"'See that bird?' he says. 'It's a Spencer's warbler. (I knew he didn't know the real name.) Well, in Italian, it's a Chutto Lapittida. In Portuguese, it's a Bom da Peida. In Chinese, it's a Chung-long-tah, and in Japanese, it's a Katano Tekeda. You can know the name of that bird in all the languages of the world, but when you're finished, you'll know absolutely nothing whatever about the bird. You'll only know about humans in different places, and what they call the bird. So let's look at the bird and see what it's doing, that's what counts.'"*

In the following sections, I will discuss basic OOP concepts in detail.

Encapsulation: Hiding the Details

Accountants love details (all the numbers, receipts, and invoices). The accountant's boss, however, is interested in the bottom line. If the bottom line is zero, the company is debt-free. If the bottom line is positive, the company is profitable. She is happy to ignore all the messy details and focus on other things. Encapsulation is about ignoring or hiding internal details. In business, this is delegation. Without it, the boss may need to deal with accounting, tax law, and international trading at a level beyond her ability.

OOP loves encapsulation. With encapsulation, classes hide their own internal details. Users of a class (yourself, other developers, or other applications) are not required to know or care why it works. Class users just need the available service names and what to provide to use them. Building classes is an abstraction process; you start with a complex problem, and then reduce it down (abstracting it) to a list of related services. Encapsulation simplifies software development and increases the potential for code reuse.

To demonstrate, I'll present some pseudo-code (false code). You can't enter pseudo-code into a computer, but it's great for previewing ideas. First, you need an accounting class:

```
Start Of Accounting Class
End Of Accounting Class
```

Everything between the start and end line is the accounting class. A useless class so far, because it's empty. Let's give the accounting class something to do:

```
Start Of Accounting Class
    Start Of Bottom Line Service
        (Internal Details Of Bottom Line Service)
    End Of Bottom Line Service
End Of Accounting Class
```

Now the accounting class has a bottom-line service. How does that service work? Well, I know (because I wrote the code), but you (as a user of my class) have no idea. That's exactly how it should be. You don't know or care how my class works. You just use the bottom-line service to see if the company is profitable. As long as my class is accurate and dependable, you can go about your business. You want to see the details anyway? Okay, here they are:

```
Start Of Accounting Class
    Start Of Bottom Line Service
        Do Invoice Service
        Do Display Answer Service
    End Of Bottom Line Service
End Of Accounting Class
```

Where did the Invoice and Display Answer services come from? They're part of the class too, but encapsulation is hiding them. Here they are:

```
Start Of Accounting Class
    Start Of Bottom Line Service
        Do Invoice Service
        Do Display Answer Service
    End Of Bottom Line Service

    Start Of Invoice Service
        (Internal Details Of Invoice Service)
    End Of Invoice Service
    Start Of Display Answer Service
        (Internal Details Of Display Answer Service)
    End Of Display Answer Service
End Of Accounting Class
```

The bottom-line service has no idea how the invoice service works, nor does it care. You don't know the details, and neither does the bottom-line service. This type of simplification is the primary benefit of encapsulation. Finally, how do you request an answer from the bottom-line service? Easy, just do this:

```
Do Bottom Line Service
```

That's all. You're happy, because you only need to deal with a single line of code. The bottom-line service (and encapsulation) handles the details for you.

NOTE *When I speak of hiding code details, I'm speaking conceptually. I don't mean to mislead you. This is just a mental tool to help you understand the importance of abstracting the details. With encapsulation, you're not actually hiding code (physically). If you were to view the full accounting class (as follows), you'd see the same code that I see.*

```
Start Of Accounting Class
    Start Of Bottom Line Service
        Do Invoice Service
        Do Display Answer Service
    End Of Bottom Line Service

    Start Of Invoice Service
        Gather Invoices
        Return Sum
    End Of Invoice Service

    Start Of Display Answer Service
        Display Sum
    End Of Display Answer Service
End Of Accounting Class
```

TIP *If you're wondering why some of the lines are indented, this is standard practice (that is not followed often enough). It shows, at a glance, the natural hierarchy of the code (of what belongs to what). Please adopt this practice when you write computer code.*

Inheritance: Avoid Rebuilding the Wheel

Grog roll wheel. Wheel good. Grog doesn't like rebuilding wheels. They're heavy, made of stone, and tend to crush feet when they fall over. Grog likes the wheel that his stone-age neighbor built last week. Sneaky Grog. Maybe he'll carve some holes into the wheel to store rocks, twigs, or a tasty snack. If Grog does this, he'll have added something new to the existing wheel (demonstrating inheritance long before the existence of computers).

Inheritance in OOP has a very nice extra. You don't need to modify your neighbor's wheel. You only need to tell the computer, "Build a replica of my neighbor's wheel, and then add this, and this, and this." The result is a custom wheel, but you didn't modify the original. Now you have two wheels, each unique. To clarify, here's some more pseudo-code:

```
Start Of Wheel Class
    Start Of Roll Service
        (Internal Details Of Roll Service)
    End Of Roll Service
End Of Wheel Class
```

The wheel class provides a single service named roll. That's a good start, but what if you want to make a tire? Do you build a new tire class from scratch? No, you just use inheritance to build a tire class, like this:

```
Start Of Tire Class
    Using Wheel Class
End Of Tire Class
```

By using the wheel class as a starting point, the tire class already knows how to roll (the tire is a type of wheel). Here's the next logical step:

```
Start Of Tire Class
    Using Wheel Class
    Property Named Size
End Of Tire Class
```

Now the tire class has a property named size. That means you could create many unique tire objects. All of the tires can roll (behavior inherited from the wheel class), but each tire has its own unique size. You could add other properties to the tire class too. With very little work, you could have small car tires that roll, big truck tires that roll, and bigger bus tires that roll.

Polymorphism: Exhibiting Similar Features

Oranges have pulp. Lemons have pulp. Grapefruits have pulp. Cut any of these open, I dare you, and try to scoop out the fruit with a spoon. Chances are, you'll get a squirt of citrus juice in your eye. Citrus fruits know exactly where your eye is, but you don't have to spoon them out to know they share this talent (they're all acid-based juice-squirters). Look at the following citrus class:

```
Start Of Citrus Class
    Property Named Juice

    Start Of Taste Service
        (Internal Details Of Taste Service)
    End Of Taste Service

    Start Of Squirt Service
        (Internal Details Of Squirt Service)
    End Of Squirt Service
End Of Citrus Class
```

You can use the citrus class as a base to define other classes:

```
Start Of Orange Class
    Using Citrus Class
    Juice is Orange
End Of Orange Class

Start Of Lemon Class
    Using Citrus Class
    Juice is Lemon
End Of Lemon Class

Start Of Grapefruit Class
    Using Citrus Class
    Juice is Grapefruit
End Of Grapefruit Class
```

Besides demonstrating inheritance again, the orange, lemon, and grapefruit classes also exhibit similar behaviors. You know that the orange, lemon, and grapefruit classes have the ability to squirt and each has a Juice property (inherited from the Citrus class). So the orange can squirt orange juice, the lemon can squirt lemon juice, and the grapefruit can squirt grapefruit juice. You don't have to know in advance which type of fruit, because they all squirt. In fact, you could taste the juice (inherited from the citrus class) to know which fruit you're dealing with. That's polymorphism: multiple objects exhibiting similar features in different ways.

What's Next?

If this was your first exposure to OOP, maybe you were expecting something more complicated. I hope I have demonstrated its simplicity instead. Coming up next, I will focus on the general programming concepts common to modern computer languages.